SI-PRÁCTICA 1

Grupo 1 - Prácticas

Lawrence Arthur Rider García

********_

Larg1@alu.ua.es

1. Introducción

Para esta práctica hemos utilizado el entorno de desarrollo 'NetBeans. Además, hemos utilizado Simbad, que es un simulador robótico en 3D desarrollado en Java.
La práctica consiste en que habrá un laberinto (un mapa), con una casilla de entrada y una salida. En primer lugar, deberemos programar en java un algoritmo A* capaz de encontrar de forma lo más óptima posible un camino hacia la salida. Por último, deberemos definir unas reglas en el controlador difuso para que el robot sea capaz de seguir dicho camino.

2. Algoritmo A*

2.1 Explicación clase Nodo

Para poder empezar a implementar este algoritmo, he tenido que crear una clase auxiliar llamada Nodo.java

Esta clase Nodo está compuesta por dos enteros X e Y (fila y columna), que será la posición de dicho nodo en el mundo. Además, también tendrán un Nodo padre que apuntará al padre de cada uno.
Como en la clase Nodo utilizo las variables origen, destino, tamaño y mundo, y éstas no cambian, decidí guardar dichas variables como estáticas e inicializarlas al principio del algoritmo A* mediante la función llamada "inicializarNodos", pasando por parámetro dichas variables.

```
/**
 *
 * @author larry
 */
public class Nodo implements Comparable<Nodo>{

    private final int x,y;
    private Nodo nodoPadre;
    private static int destino, tamaño, mundo[][];

    public Nodo(int x, int y, Nodo nodoPadre) {
        this.x = x;
        this.y = y;
        this.nodoPadre = nodoPadre;
    }
```

A la hora de crear un Nodo, almaceno su posición “X” e “Y”, y el padre.

```
public Nodo(int x, int y, Nodo nodoPadre) {
    this.x = x;
    this.y = y;
    this.nodoPadre = nodoPadre;
}
```

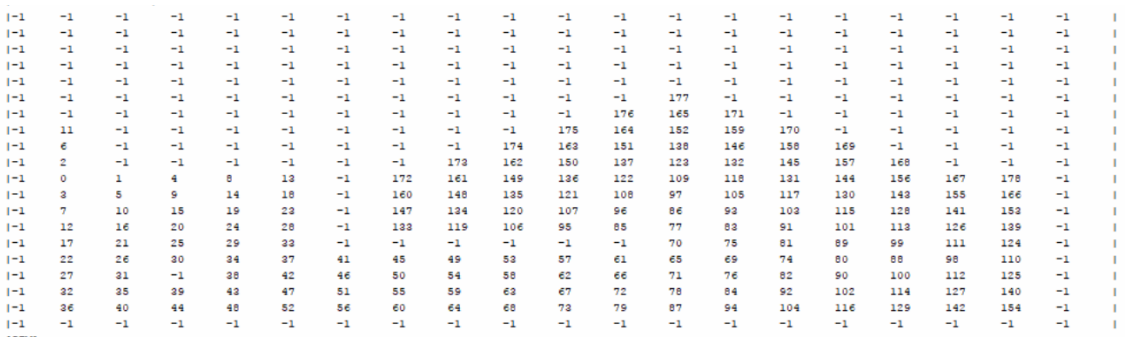
La clase Nodo obtiene su función G mediante “getFuncionG()”:

```
private Integer getFuncionG(){
    Integer g = 0;
    if(getNodoPadre() != null){
        g = 1 + getNodoPadre().getFuncionG();
    }
    return g;
}
```

La función G es sencilla de calcular y no me dio ningún problema, es simplemente la distancia recorrida que lleva el padre, pero sumándole uno que es la distancia entre padre e hijo.

```
public Integer getFuncionH(){
    return (int) Math.abs(destino-x) + Math.abs((tamaño-1)-y); //distancia manhattan
}
```

A la hora de calcular la función H al principio lo hice mal, y resulta que me calculaba para todos los nodos el mismo valor de H. Esto supuso que a la hora de calcular el camino explorara una mayor cantidad de nodos de los que en realidad debería explorar.

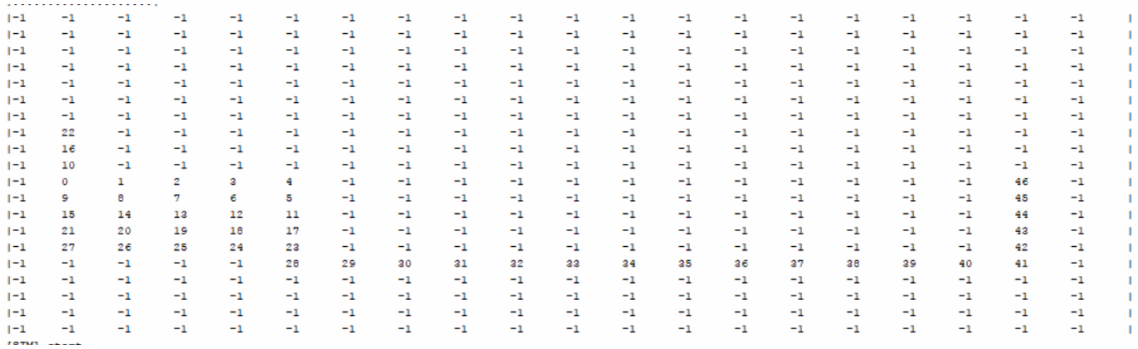


Esto sucedía porque a la hora de explorar un nodo, siempre cogía el más prometedor, y había muchos que eran igual de prometedores.

La función en la que yo decido cual es el nodo más prometedor es la siguiente:

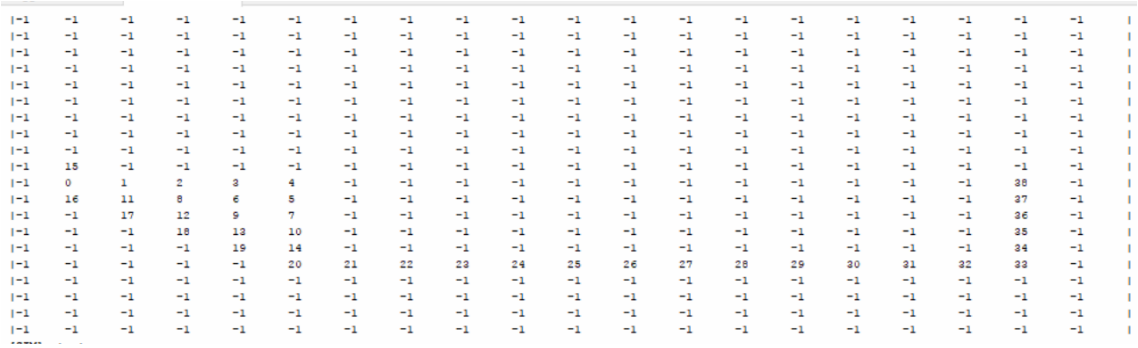
```
@Override
public int compareTo(Nodo nodo) {
    if(getFuncionF().compareTo(nodo.getFuncionF()) == 0){
        if(getFuncionH().compareTo(nodo.getFuncionH()) == 0){
            return getFuncionG().compareTo(nodo.getFuncionG());
        } else{
            return getFuncionH().compareTo(nodo.getFuncionH());
        }
    } else{
        return getFuncionF().compareTo(nodo.getFuncionF());
    }
}
```

Esta función lo que hace es comparar un nodo con otro para que el array de nodos pueda ser ordenado correctamente. Primero comparo la función F, si tuvieran la misma entonces miraría la mejor H y, por último, si fueran iguales se compararía la G. El resultado con este orden de comparación es el siguiente.



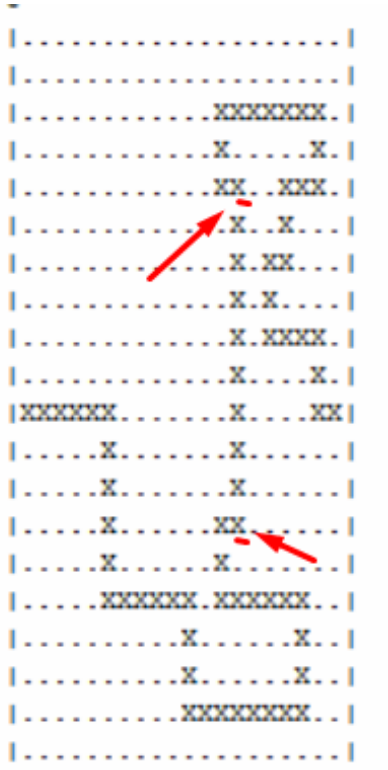
Lo que da un total de 46 nodos expandidos.

Probando me di cuenta de que, si comparamos primero la H, luego la F, y por último la G, los nodos expandidos son los siguientes (38):

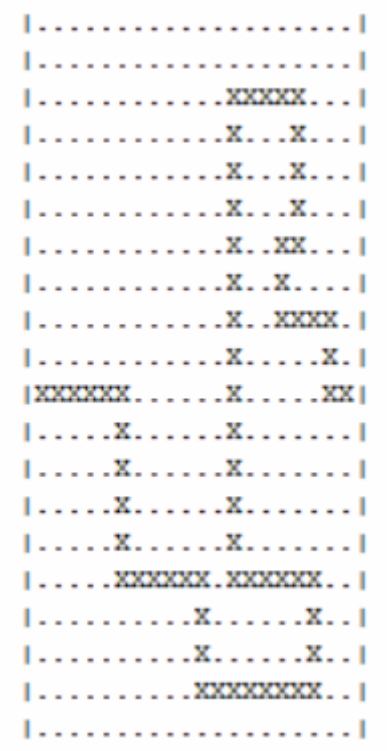


Podemos observar entonces que expande bastante menos nodos que comparando primero la función F. *(en otros mapas da mejor resultado)

Sin embargo, no siempre nos dará el camino más óptimo, ya que por ejemplo en este otro mapa, provoca que el camino sea dos nodos más largo, puesto que expande dos nodos que realmente están más cerca de la solución, pero sin tener en cuenta que hay un muro.



El camino hace dos nodos más, en vez de seguir recto como en la imagen derecha en la que se compara primero la función F.



En cuanto a la función “getFuncionF()” lo único que cabe destacar es que devuelve la suma de “getFuncionH()” y “getFuncionG()”.

```
public Integer getFuncionF(){
    return getFuncionG() + getFuncionH();
}
```

Para trabajar con el Nodo padre hay dos funciones. “getNodoPadre()” que devuelve su valor y “setNodoPadre(Nodo padre)” que actualiza el padre.

En una parte necesité saber si un nodo era el mismo que otro, para esto decidí simplemente comparar sus componentes X e Y, ya que las funciones pueden cambiar dependiendo de por donde se haya llegado al mismo nodo.

```
public boolean esIgual(Nodo nodo) {
    return (x == nodo.getX() && y == nodo.getY());
}
```

Para saber cuándo un Nodo es el Nodo final:

```
public Boolean esMeta(){
    return (x == destino && y == tamaño-2);
}
```

Como aclaración hay que decir que es -2 porque en -1 (aunque sea la salida), hay una pared, y en mi código un nodo nunca va a ser creado en una pared.

En el algoritmo A* hay un momento en el que hace falta coger los hijos de un nodo que no estén en una lista. Para ello está esta función.

```
public ArrayList<Nodo> getHijosNoAñadidosEn(ArrayList<Nodo> lista){ //Que no esten ya en la lista
    ArrayList<Nodo> hijos = new ArrayList<Nodo>();
    if (mundo[getX()][getY() + 1] == 0){ //derecha
        hijos.add(new Nodo(getX(), getY() + 1, this));
    }
    if (mundo[getX()][getY() - 1] == 0){ //izquierda
        hijos.add(new Nodo(getX(), getY() - 1, this));
    }
    if (mundo[getX() + 1][getY()] == 0){ //abajo
        hijos.add(new Nodo(getX() + 1, getY(), this));
    }
    if (mundo[getX() - 1][getY()] == 0){ //arriba
        hijos.add(new Nodo(getX() - 1, getY(), this));
    }
    for(int i = 0; i < hijos.size(); i++){
        if(hijos.get(i).estaEn(lista)){
            hijos.remove(i);
            i--;
        }
    }
    return hijos;
}
```

Empieza añadiendo al arraylist “hijos” aquellos que sean hijos válidos del nodo que llama a la función, creándolos según sus respectivas X e Y, y estableciendo como padre dicho nodo. Por último, para devolver solo aquellos que no estén ya en la lista pasada por parámetro, se eliminan de “hijos” aquellos que ya estén en la lista. Para saber si están o no, se utiliza la función “estaEn(lista)”.

```
public Boolean estaEn(ArrayList<Nodo> lista){
    for(Nodo n: lista){
        if(this.esIgual(n)){
            return true;
        }
    }
    return false;
}
```

Y ya por último solo queda la función “getPosicionEn(Lista)” la cual devuelve la posición del nodo que la llama en dicha lista.

```
public Integer getPosicionEn(ArrayList<Nodo> lista){
    for(int i=0; i<lista.size(); i++){
        if(this.esIgual(lista.get(i))){
            return i;
        }
    }
    return -1;
}
```

2.2 Comparación de heurísticas

Si utilizamos una heurística euclídea, obtenemos este resultado:

```
public Integer getFuncionH(){
    //return (int) Math.abs(destino-x) + Math.abs((tamaño-1)-y); //distancia manhattan
    return (int) Math.round(Math.sqrt(Math.pow((destino-x), 2) + Math.pow(((tamaño-1)-y), 2)));
}
```

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
22	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
16	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
0	1	2	3	4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	98
9	8	7	6	5	-1	-1	-1	-1	-1	-1	89	87	86	84	83	81	78
15	14	13	12	11	-1	-1	-1	-1	-1	-1	66	63	61	59	82	80	77
21	20	19	18	17	-1	-1	-1	-1	-1	-1	65	62	60	58	56	79	76
25	24	23	27	26	-1	-1	-1	-1	-1	-1	45	44	43	57	55	54	75
38	37	36	35	28	29	30	31	32	33	34	39	40	41	42	52	53	85
51	50	-1	49	48	47	46	71	70	69	68	67	64	90	88	-1	-1	-1
74	72	73	97	96	95	94	93	92	91	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Es cierto que encuentra un camino válido igualmente, pero para ello expande 98 nodos, lo que supone prácticamente el doble que por ejemplo en manhattan.

```
public Integer getFuncionH(){
    return (int) Math.abs(destino-x) + Math.abs((tamaño-1)-y); //distancia manhattan
}
```

22	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
16	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
0	1	2	3	4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	46
9	8	7	6	5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	45
15	14	13	12	11	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	44
21	20	19	18	17	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	43
27	26	25	24	23	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	42
-1	-1	-1	-1	28	29	30	31	32	33	34	35	36	37	38	39	40	41
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Por lo tanto, podemos determinar que calcular la heurística con la distancia Manhattan consigue mejores resultados. Esto es completamente normal dado que el camino natural del robot no es diagonal y sí es como se calcula la de Manhattan.

2.2 Explicación algoritmo A* implementado

Para empezar, ¿Qué es un algoritmo A*? Es un algoritmo de búsqueda heurística que, por lo tanto, utiliza una función heurística. Ésta nos da un valor para cada celda. La función heurística es una estimación optimista de cómo de lejos está el objetivo.

En este algoritmo tendremos dos listas: "listaInterior" y "listaFrontera". En "listaInterior" se irán almacenando aquellos nodos que ya hayan sido expandidos y, por lo tanto, no deberían de volver a ser explorados. Y en "listaFrontera" se almacenarán aquellos nodos que se van a explorar.

Por lo tanto, en el instante inicial, habrá que meter en "listaFrontera" el nodo inicial y "listaInterior" deberá de estar vacía.

```
//Calcula el A*
public int AEstrella(){
    int result = 0;

    Nodo.inicializarNodos(destino, tamaño, mundo);
    ArrayList<Nodo> listaInterior = new ArrayList<Nodo>();
    ArrayList<Nodo> listaFrontera = new ArrayList<Nodo>();
    listaFrontera.add(new Nodo(origen, 1, null));
    int contador = 0;

    while(!listaFrontera.isEmpty()){
```

En la anterior foto se puede observar como almacenamos las variables "destino", "tamaño" y "mundo" en la clase Nodo, y como añadimos el nodo origen (sin padre porque es el inicial) en "listaFrontera".

Mientras "listaFrontera" no esté vacía, es decir, mientras aún queden nodos por explorar, habrá que seguir buscando.
Para empezar, si no está vacía, hay que coger el nodo más prometedor de "listaFrontera", ya que es el que utilizaremos para analizar sus hijos (y así sucesivamente) hasta dar con la salida.

```
while(!listaFrontera.isEmpty()){
    Collections.sort(listaFrontera);
    Nodo nodo = listaFrontera.get(0);
    expandidos[nodo.getX()][nodo.getY()] = contador;
    contador++;
    if(nodo.esMeta()){
        //camino[destino][tamaño-1] = 'X'; //pone una X en el nodo final
        //camino[origen][0] = 'X'; //pone una X en el nodo inicial
        construirCamino(nodo);
        break;
    }
```

Para coger dicho nodo prometedor lo que hago es ordenar la "listaFrontera" y coger el primero de la lista. El criterio de ordenación ha sido explicado anteriormente en la clase Nodo.java, en la función "compareTo".

Además, también se puede observar que el array "expandidos" lo voy actualizando con un contador simple. Lo único que hago es guardar su valor en dicho array en la posición del nodo que acabamos de seleccionar como prometedor.

Antes de continuar con la ejecución, hay que analizar si este nodo es ya la meta, en caso positivo deberemos de acabar con el bucle y construir el camino a partir de dicho nodo. Esto lo hago con la siguiente función recursiva:

```
private void construirCamino(Nodo nodo){
    camino[nodo.getX()][nodo.getY()] = 'X';
    if(nodo.getNodoPadre() != null){
        construirCamino(nodo.getNodoPadre());
    }
}
```

La función recursiva acabará cuando el nodo deje de tener padre, es decir, que sea el nodo inicial.

Si el nodo no es meta, entonces hay que quitarlo de "listaFrontera" porque vamos a analizarlo, y por tanto hay que meterlo en "listaInterior".
Una vez hecho esto, hay que recoger todos sus hijos e ir analizando uno a uno. Eso sí, solo hay que coger aquellos hijos que no estén ya añadidos en "listaInterior", es decir, solo aquellos que no hayan sido ya expandidos.

```
listaFrontera.remove(nodo);
listaInterior.add(nodo);

ArrayList<Nodo> hijos = nodo.getHijosNoAñadidosEn(listaInterior);

for(int i=0; i<hijos.size(); i++){
    Nodo hijo = hijos.get(i);
    if(!hijo.estaEn(listaFrontera)){
        //hijo.setNodoPadre(nodo); //ya lo hago al crear los hijos
        listaFrontera.add(hijo);
    } else{
        Integer posicion = hijo.getPosicionEn(listaFrontera);
        if(posicion != -1 && hijo.getFuncionG() <= listaFrontera.get(posicion).getFuncionG()){
            listaFrontera.get(posicion).setNodoPadre(nodo);
        }
    }
}
```

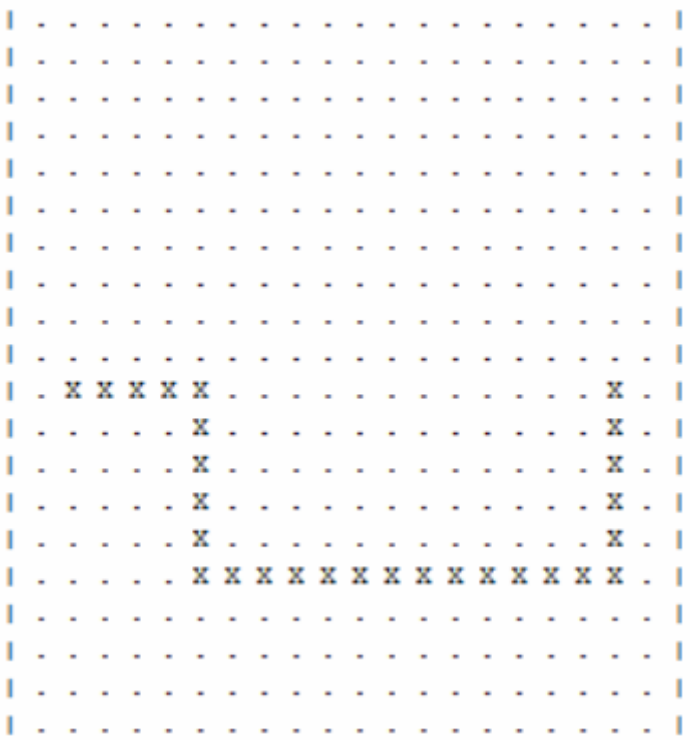
Para cada hijo del nodo, comprobamos si estaba ya añadido en "listaFrontera".

Si no estaba añadido ya, simplemente lo añado a “listaFrontera” como nuevo nodo descubierto que hay que explorar.

Si ya lo estaba quiere decir que hemos llegado al mismo nodo a través de otro padre y, por lo tanto, tenemos que ver si hemos llegado por un camino mejor o no. Si se llega antes a ese nodo por el nuevo camino, hay que actualizar su padre para que lo sea ahora el óptimo. Si por el contrario, se llega a ese hijo por un camino peor, no hay que hacer nada (puesto que ya está en “listaFrontera” y ya se analizará más adelante si fuera necesario).

Para terminar con el algoritmo, simplemente imprimo los vectores “expandidos” y “camino”.

Solución para el mapa dado en la práctica:



Nodos expandidos para hallar dicha solución:

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
22	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
16	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
0	1	2	3	4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	46
9	8	7	6	5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	45
15	14	13	12	11	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	44
21	20	19	18	17	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	43
27	26	25	24	23	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	42
-1	-1	-1	-1	28	29	30	31	32	33	34	35	36	37	38	39	40	41
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

3. Diseño SE difuso

Para esta parte de la práctica, debemos realizar un sistema experto difuso que consiga que el robot siga el camino realizado por el algoritmo A*, evitando obstáculos utilizando los sensores y actuadores del robot.

Para ello vamos a utilizar el lenguaje FCL basado en la transcripción del lenguaje natural para especificar sistemas difusos.

El robot tiene 9 sensores, 8 que representan lecturas del robot (determinan la cercanía a las paredes) y el sensor 'sig' que indica cuantos grados debe girar el robot para dirigirse a la X mas cercana.

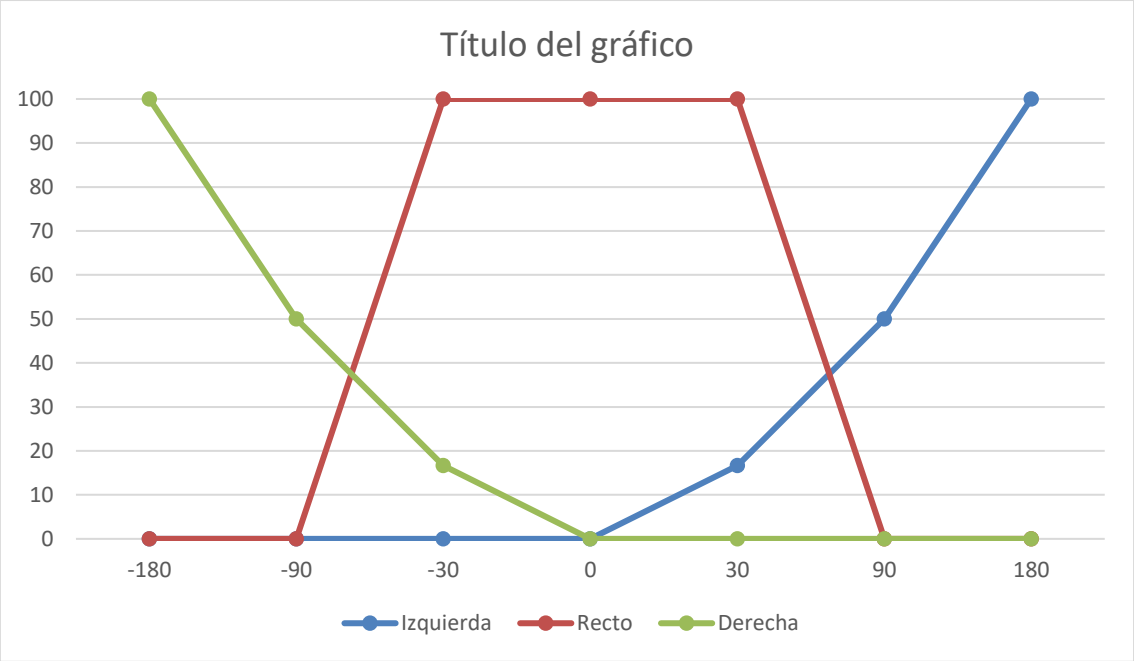
```
//EJEMPLO de definición NO VINCULANTE. Hacer lo mismo para los 7 sensores restantes
FUZZIFY s0
    TERM near   := (0,1) (0.5,0.5) (1,0.25) (1.5,0);
    TERM med    := (0.25,0) (0.75,1) (1.25,0) (1.5,0);
    TERM far    := (0,0) (1.5,1);
END_FUZZIFY

FUZZIFY s1
    TERM near   := (0,1) (0.5,0.5) (1,0.25) (1.5,0);
    TERM med    := (0.25,0) (0.75,1) (1.25,0) (1.5,0);
    TERM far    := (0,0) (1.5,1);
END_FUZZIFY

FUZZIFY s2
    TERM near   := (0,1) (0.5,0.5) (1,0.25) (1.5,0);
    TERM med    := (0.25,0) (0.75,1) (1.25,0) (1.5,0);
    TERM far    := (0,0) (1.5,1);
END_FUZZIFY
```

El resto de definiciones hasta el sensor 's8' es igual.

La definición del sensor 'sig' es la siguiente.

```
FUZZIFY sig
    TERM izquierdam :=    (-180,0) (0,0) (180,1);
    TERM izquierda  :=    (-180,0) (-90,0) (0,1) (90,1) (180,0);
    TERM recto      :=    (-180,0) (-90,0) (-30,1) (30,1) (90,0) (180,0);
    TERM derecha    :=    (-180,0) (-90,1) (0,1) (90,0) (180,0);
    TERM derecham   :=    (-180,1) (0,0) (180,0);
END_FUZZIFY
```

Para intentar realizar la definición del sensor ‘sig’ lo más correcta posible me hice la siguiente gráfica.



Más tarde definí también las variables izquierdam y derecham, las cuales hacen un giro más brusco.
Tanto ‘vel’ como ‘rot’ ya estaban definidas y en principio no hace falta modificarlas.

Por último, lo que quedaría por hacer es definir las reglas necesarias para que el robot siga el camino.
Yo lo que he hecho ha sido primero ponerle la velocidad en función del sensor0, y luego definir que si tanto el s1, como el s2 y el s3 detectan una pared que gire a la derecha. Y si el s6, s7 y s8 detectan también una pared que gire a la izquierda. Son normas simplemente para que no choque.

```
RULE 0: IF s0 IS near THEN vel is slow;

RULE 1: IF s1 IS near THEN rot IS derm;
RULE 2: IF s1 IS med THEN rot IS derm;
RULE 3: IF s2 IS near THEN rot IS derm;
RULE 4: IF s2 IS med THEN rot IS derm;
RULE 5: IF s3 IS near THEN rot IS derm;
RULE 6: IF s3 IS med THEN rot IS derm;

RULE 7: IF s8 IS near THEN rot IS izqm;
RULE 8: IF s8 IS med THEN rot IS izqm;
RULE 9: IF s7 IS near THEN rot IS izqm;
RULE 10: IF s7 IS med THEN rot IS izqm;
RULE 11: IF s6 IS near THEN rot IS izqm;
RULE 12: IF s6 IS med THEN rot IS izqm;
```

Y ya para terminar me quedaba definir las reglas para que el robot siguiera el camino en base a el sensor ‘sig’.

```
RULE 13: IF sig IS recto THEN rot IS cen;
RULE 14: IF sig IS izquierdam THEN rot IS izqm;
RULE 15: IF sig IS derecha THEN rot IS der;
RULE 16: IF sig IS derecham THEN rot IS derm;
```

A la hora de determinar estas últimas reglas tuve un problema y es que si definía por ejemplo la regla que si detecta “izquierda” rote “izq”, entraba en *conflicto* con la regla 15 y el robot se chocaba con la pared. Así que tuve que descartarla y dejarlo así para que pudiera llegar a la salida en el mapa dado.